Introduction to Operating Systems

What's an OS?

The OS is a layer between applications and hardware to ease development.

- **Abstraction**. provides abstraction for applications:
  - manages + hides hardware details
  - uses low-level interfaces (not available to applications)
  - multiplexes hardware to multiple programs (*virtualization*)
  - makes hardware use efficient for applications
- **Protection**.
  - from processes using up all resources (*accounting*, *allocation*)
  - from processes writing into other processes memory
- **Resource Management**.
  - manages + multiplexes hardware resources
  - decides between conflicting requests for resource use
  - *goal*: efficient + fair resource use
- **Control**.
  - controls program execution
  - prevents errors and improper computer use

⇝ **no universally accepted definition**

Hardware Overview

- **Bus**: CPU(s)/devices/memory (conceptually) connected to common bus
  - CPU(s)/devices competing for memory cycles/bus
  - all entities run concurrently
  - *today*: multiple buses
- **Device controller**: has local buffer and is in charge of particular device
- **Interplay**:
  1. CPU issues commands, moves data to devices
  2. Device controller informs APIC that it has finished operation
  3. APIC signals CPU
  4. CPU receives device/interrupt number from APIC, executes handler

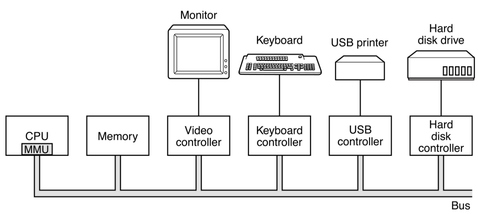


Figure 1: Traditional bus design.

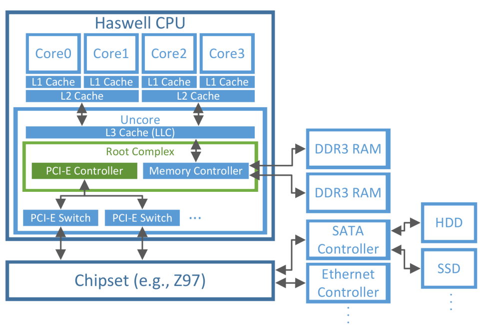


Figure 2: Modern bus design.

Central Processing Unit (CPU) — Operation

- **Principle**:
  1. *fetches* instructions from memory,
  2. *executes* them
- **During execution**: (meta-)data is stored in CPU-internal registers, i.e.
  - general purpose registers
  - floating point registers
  - instruction pointer (IP)
  - stack pointer (SP)
  - program status word (PSW)

CPU — Modes of Execution

- **User mode** (x86: *Ring 3/CPL 3*):
  - only non-privileged instructions may be executed
  - cannot manage hardware → *protection*
- **Kernel mode** (x86: *Ring 0/CPL 0*):
  - all instructions allowed
  - can manage hardware with *privileged instructions*

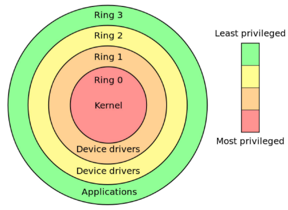


Figure 3: The different protection layers in the *ring model*.

Random Access Memory (RAM)

- **Principle**: keeps currently executed instructions + data
- **Connectivity**:
  - *today*: CPUs have built-in **memory controller**
  - *CPU caches*: "wired" to CPU
  - *RAM*: connected via pins
  - *PCI-E switches*: connected via pins

Caches

- **Problem**: RAM delivers instructions/data slower than CPU can execute
- **Locality principle**:
  - *spatial locality*: future refs often near previous accesses (e.g. next byte in array)
  - *temporal locality*: future refs often at previously accessed ref (e.g. loop counter)
- **Solution**: *caching* helps mitigating this memory wall
  1. *copy* used information temporarily from slower to faster storage
  2. *check* faster storage first before going down *memory hierarchy*
  3. if *not found*, data is copied to cache and used from there
- **Access latency**:
  - *register*: ~1 CPU cycle
  - *L1 cache* (per core): ~4 CPU cycles
  - *L2 cache* (per core pair): ~12 CPU cycles
  - *L3 cache/LLC* (per uncore): ~28 CPU cycles (~25 GiB/s)
  - *DDR3-12800U RAM*: ~28 CPU cycles + ~ 50ns (~12 GiB/s)

Device controlling

- **Device controller**: controls device, accepts commands from OS via *device driver*
- **Device registers/memory**:
  - *control* device by writing device registers
  - *read* status of device by reading device registers
  - *pass data* to device by reading/writing device memory
- **Device registers/memory access**:
  1. **port-mapped IO** (PMIO): use special CPU instructions to access port-mapped registers/memory
  2. **memory-mapped IO** (MMIO):
     - use same address space for RAM and device memory
     - some addresses map to RAM, others to different devices
     - access device's memory region to access device registers/memory
  3. **Hybrid**: some devices use hybrid approaches using both

Summary

- The OS is an **abstraction** layer between applications and hardware (multiplexes hardware, hides hardware details, provides protection between processes/users)
- The CPU provides a **separation** of User and Kernel mode (which are required for an OS to provide protection between applications)
- CPU can execute commands faster than memory can deliver instructions/data — memory **hierarchy** mitigates this memory wall, needs to be carefully managed by OS to minimize slowdowns
- device drivers **control** hardware devices through PMIO/MMIO
- Devices can **signal** the CPU (and through the CPU notify the OS) through interrupts

OS Concepts

OS Invocation

- OS Kernel does **not** always run in background!
- Occasions invoking kernel, switching to kernel mode:
  1. **System calls**: User-Mode processes require higher privileges
  2. **Interrupts**: CPU-external device sends signal
  3. **Exceptions**: CPU signals unexpected condition

System Calls — Motivation

- **Problem**: protect processes from one another
- **Idea**: Restrict processes by running them in user-mode
- ⇝ **Problem**: now processes cannot manage hardware,...
  - who can switch between processes?
  - who decides if process may open certain file?
- ⇝ **Idea**: OS provides **services** to apps
  1. app calls system if service is needed (**syscall**)
  2. OS checks if app is allowed to perform action
  3. if app may perform action and hasn't exceeded quota, OS performs action in behalf of app in kernel mode

System Calls — Examples

- `fd = open(file, how,...)` – open file for read/write/both
- documented e.g. in `man 2 write`
- overview in `man 2 syscalls`

System Calls vs. APIs

- **Syscalls**: interface between apps and OS services, limited number of well-defined entry points to kernel
- **APIs**: often used by programmers to make syscalls (e.g. `printf` library call uses `write` syscall)
- common APIs: Win32, POSIX, C API

System Calls — Implementation

- **Trap Instruction**: single syscall interface (entry point) to kernel
  - switches CPU to kernel mode, enters kernel in same way for all syscalls
  - *system call dispatcher* in kernel then acts as syscall multiplexer
- **Syscall Identification**: number passed to trap instruction
  - *Syscall Table* maps syscall numbers to kernel functions
  - *Dispatcher* decides where to jump based on number and table
  - programs (e.g. `stdlib`) have syscall number compiled in!

  ⇝ never reuse old syscall numbers in future kernel versions

Interrupts

- **Devices**: use interrupts to signal predefined conditions to OS
  - *reminder*: device has "interrupt line" to CPU (e.g. device controller informs CPU that operation is finished)
- **Programmable Interrupt Controller**: manages interrupts
  - interrupts can be *masked* (queued, delivered when interrupt unmasked)
  - queue has finite length ⇝ interrupts can get lost
- **Examples**:
  1. *timer-interrupt*: periodically interrupts processes, switches to kernel ⇝ can then switch to different processes for fairness
  2. *network interface card* interrupts CPU when packet was received ⇝ can deliver packet to process and free NIC buffer
- **Interrupt process**:
  1. CPU looks up *interrupt vector* (= table pinned in memory, contains addresses of all service routines)
  2. CPU transfers control to respective *interrupt service routine* in OS that handles interrupt

  ⇝ interrupt service routine must first save interrupted process's state (instruction pointer, stack pointer, status word)

Exceptions

- **Motivation**: unusual condition → impossible for CPU to continue processing
- ⇝ **Exception** generated within CPU:
  1. CPU interrupts program, gives kernel control
  2. kernel determines reason for exception
  3. if kernel can resolve problem ⇝ does so, continues *faulting instruction*
  4. kills process if not
- **Difference to Interrupts**: interrupts can happen in any context, exceptions always occur asynchronous and in process context

OS Concepts — Physical Memory

- up to early 60s:
  - programs loaded and run directly in *physical memory*
  - program too large → partitioned manually into *overlays*
  - OS: swaps overlays between disk and memory
  - different jobs could observe/modify each other

OS Concepts — Address Spaces

- **Motivation**: bad programs/people need to be isolated
- **Idea**: give every job the illusion of having all memory to itself
  - every job has own *address space*, can't name addresses of others
  - jobs always and only use virtual addresses

Virtual Memory — Indirect Addressing

- **MMU**: every CPU has built-in *memory management unit* (MMU)
- **Principle**: translates virtual addresses to physical addresses at every load/store

  ⇝ address translation protects one program from another
- **Definitions**:
  - *Virtual address*: address in process' address space
  - *Physical address*: address of real memory

Virtual Memory — Memory Protection

- **Kernel-only Virtual Addresses**
  - kernel typically part of all address spaces
  - ensures that apps can't touch kernel memory
- **Read-only virtual addresses**: can be enforced by MMU
  - allows safe sharing of memory between apps
- **Execute Disable**: can be enforced by MMU
  - makes code injection attacks harder

Virtual Memory — Page Faults

- **Motivation**: not all addresses need to be mapped at all times
  - MMU issues *page fault* exception when accessed virtual address isn't mapped
  - OS handles page faults by loading faulting addresses and then continuing the program

  ⇝ memory can be *over-committed*: more memory than physically available can be allocated to application
- **Illegal addresses**: page faults also issued by MMU on illegal memory accesses

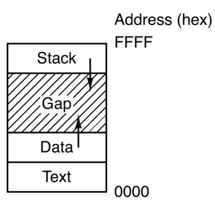
OS Concepts — Processes

- **Process**: program in execution ("instance" of program)
- each process is associated with
  - **Process Control Block** (PCB): contains information about allocated resources
  - virtual **Address Space** (AS):
    - all (virtual) memory locations a program can name
    - starts at 0 and runs up to a maximum
    - address 123 in AS1 generally ≠ address 123 in AS2
    - indirect addressing ⇝ different ASes to different programs

    ⇝ *protection between processes*

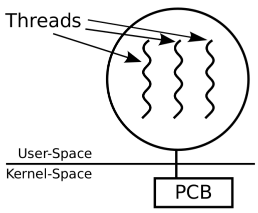
OS Concepts — Address Space Layout

- **Sections**: address spaces typically laid-out in different sections
  - memory addresses between sections *illegal*
  - illegal addresses ⇝ page fault (*segmentation fault*)
  - OS usually kills process causing segmentation fault
- **Important sections**:
  - *Stack*: function history, local variables
  - *Data*: Constants, static/global variables, strings
  - *Text*: Program code



OS Concepts — Threads

- **Thread**: represents execution state of process (≥ 1 thread per process)
  - *IP*: stores currently executed instruction (address in `text` section)
  - *SP*: stores address of stack top (> 1 threads → multiple stacks!)
  - *PSW*: contains flags about execution history (e.g. last calculation was 0 → used in following jump instruction)
  - more general purpose registers, floating point registers,...



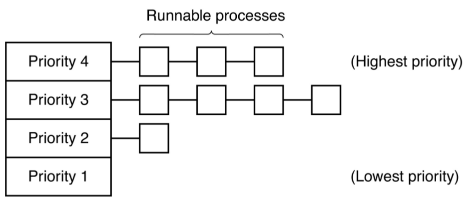
OS Concepts — Policies vs. Mechanisms

- **Mechanism**: implementation of what is done (e.g. commands to write to HDD)
- **Policy**: rules which decide when what is done and how much (e.g. how often, how many resources are used,...)

→ *mechanisms can be reused even when policy changes*

OS Concepts — Scheduling

- **Motivation**: multiple processes/threads available ⇝ OS needs to switch between them (for multitasking)
- **Scheduler**: decides which job to run next (*policy*) — tries to
  - provide fairness
  - meet performance goals
  - adhere to priorities
- **Dispatcher**: performs task-switching (*mechanism*)



OS Concepts — Files

- **Motivation**: OS hides peculiarities of file storage, programmer uses device-independent *files/directories*
- **Files**: associate *file name* and *offset* with bytes
- **Directories**: associate *directory names* with directory names or file names
- **File System**: ordered block collection
  - main task: translate (dir name + file name + offset) to block
  - programmer uses file system operations to operate on files (`open`, `read`, `seek`)
  - processes can communicate directly through special *named pipe* file (used with same operations as any other file)

OS Concepts — Directory Tree

- **Directories**: form *directory tree/file hierarchy* → structure data
- **Root Directory**: topmost directory in tree
- **Path Name**: used to specify file

OS Concepts — Mounting

- **Unix**: common to orchestrate multiple file systems in single file hierarchy
- file systems can be *mounted* on directory
- **Win**: manage multiple directory hierarchies with drive letters (e.g. `C:\Users`)

OS Concepts — Storage Management

- **OS**: provides uniform view of information storage to file systems
  - *Drivers*: hide specific hardware devices → hides device peculiarities
  - general interface abstracts physical properties to logical units → block
- **Performance**: OS increases I/O performance:
  - *Buffering*: Store data temporarily while transferred
  - *Caching*: Store data parts in faster storage
  - *Spooling*: Overlap one job's output with other job's input

Summary

- **OS**: provides abstractions for and protection between applications
- **Kernel**: does not always run — certain events invoke kernel
  - *syscall*: process asks kernel for service
  - *interrupt*: device sends signal that OS has to handle
  - *exception*: CPU encounters unusual situation
- **Processes**: encapsulate resources needed to run program in OS
  - *threads*: represent different execution states of process
  - *address space*: all memory process can name
  - *resources*: allocated resources, e.g., open files
- **Scheduler** decides which process to run next when multi-tasking
- **Virtual Memory** implements address spaces, provides protection between processes
- **File system** abstracts background store using I/O drivers, provides simple interface (files + directories)

Processes

The Process Abstraction

- **Motivation**: computers (seem to) do "several things at the same time" (quick process switching → *multiprogramming*)
- **Model**: *process abstraction* models this concurrency:
  - container contains information about program execution
  - conceptually, every progress has own "'virtual CPU"'
  - execution context is changed on process switch
  - dispatcher switches context when switching processes
  - **context switch**: dispatcher saves current registers/memory mappings, restores those of next process

Process-Cooking Analogy

- Program/Process like Recipe/Cooking
- **Recipe**: lists ingredients, gives algorithm what to do when

  ⇝ program describes memory layout/CPU instructions
- **Cooking**: activity of using the recipe

  ⇝ process is activity of executing a program
- multiple similar recipes for same dish

  ⇝ multiple programs may solve same problem
- recipe can be cooked in different kitchens at the same time

  ⇝ program can be run on different CPUs at the same time (as different processes)
- multiple people can cook one recipe

  ⇝ one process can have several worker threads

Concurrency vs. Parallelism

- OS uses currency + parallelism to implement multiprogramming
  1. **Concurrency**: multiple processes, one CPU

     ⇝ not at the same time
  2. **Parallelism**: multiple processes, multiple CPU

     ⇝ at the same time

Virtual Memory Abstraction — Address Spaces

- every process has own *virtual addresses* (`vaddr`)
- MMU relocates each load/store to *physical memory* (`pmem`)
- processes never see physical memory, can't access it directly

\+ MMU can enforce protection (mappings in kernel mode)

\+ programs can see more memory than available
  - 80:20 rule: 80% of process memory idle, 20% active
  - can keep working set in RAM, rest on disk

\- need special MMU hardware

Address Space (Process View)

- **Motivation**: code/data/state need to be organized within process

  ⇝ *address space layout*
- **Data types**:
  1. *fixed size* data items
  2. data naturally *freed in reverse allocation order*
  3. data *allocated/freed "'randomly"'*
- compiler/architecture determine how large int is and what instructions are used in text section (`code`)

- **Loader** determines based on exe file how executed program is placed in memory

Segments — Fixed-Size Data + Code

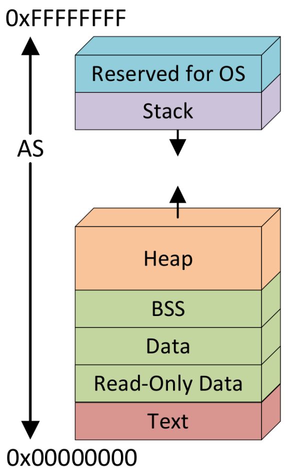
- some data in programs never changes or will be written but never grows/shrinks
  ⇝ memory can be statically allocated on process creation
- **BSS segment** (*block started by symbol*):
  - statically allocated variables/non-initialized variables
  - executable file typically contains starting address + size of BSS
  - entire segment initially 0
- **Data segment**: fixed-size, initialized data elements (e.g. global variables)
- **Read-only data segment**: constant numbers, strings
- All three sometimes summarized as one segment
- compiler and OS decide ultimately where to place which data/how many segments exist

Segments — Stack

- some data naturally freed in reverse allocation order
  - very easy memory management (stack grows upwards)
- fixed segment starting point
- store top of latest allocation in **stack pointer** (SP) (initialized to starting point)
- *allocate* `a` byte data structure: `SP += a; return(SP - a)`
- *free* `a` byte data structure: `SP -= a`

Segments — Heap (Dynamic Memory Allocation)

- some data "randomly" allocated/freed
- two-tier memory allocation:
  1. allocate large memory chunk (**heap segment**) from OS
     - base address + **break pointer** (BRK)
     - process can get more/give back memory from/to OS
  2. dynamically partition chunk into smaller allocations
     - `malloc`/`free` can be used in random oder
     - purely user-space, no need to contact kernel



Summary

Processes: recipe vs. cooking = program vs. process
- processes = resource container for OS
- process feels alone (has own CPU and memory)
- OS implements multiprogramming through rapid process switching

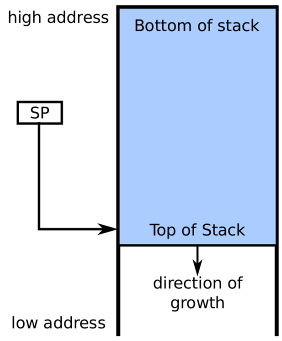
Process API

Execution Model — Assembler (simplified)

- **Principle**: OS interacts directly with compiled programs
  - switch between processes/threads ⇝ *save/restore* state
  - deal with/pass on *signals/exceptions*
  - receive *requests* from applications
- **Instructions**:
  - `mov`: Copy referenced data from second operand to first operand
  - `add`/`sub`/`mul`/`div`: Add,…from second operand to first operand
  - `inc`/`dec`: increment/decrement register/memory location
  - `shl`/`shr`: shift first operand left/right by amount given by second operand
  - `and`/`or`/`xor`: calculate bitwise and,…of two operands storing result in first
  - `not`: bitwise negate operand

Execution Model — Stack (x86)

- **stack pointer** (SP): holds address of stack top (growing downwards)
- **stack frames**: larger stack chunks
- **base pointer** (BP): used to organize stack frames



Execution Model — jump/branch/call commands (x86)

- `jmp`: continue execution at operand address
- `j$condition`: jump depending on PSW content
  - *true* ⇝ jump
  - *false* ⇝ continue
  - examples: `je` (jump equal), `jz` (jump zero)
- `call`: push function to stack and jump to it
- `return`: return from function (jump to return address)

Execution Model — Application Binary Interface (ABI)

- **Idea**: standardizes binary interface between programs, modules, OS:
  - executable/object file layout
  - calling conventions
  - alignment rules
- **calling conventions**: standardize exact way function calls are implemented
  ⇝ interoperability between compilers

Execution Model — calling conventions (x86)

- function call — **caller**:
  1. save local scope state
  2. set up parameters where function can find them
  3. transfer control flow
- function call — **called function**:
  1. set up new local scope (local variables)
  2. perform duty
  3. put return value where caller can find it
  4. jump back to caller (IP)

Passing parameters to the system

- parameters are passed through **system calls**
- call number + specific parameters must be passed
- parameters can be transferred through
  - **CPU registers** (~6)
  - **Main Memory** (heap/stack – more parameters, data types)
- ABI specifies how to pass parameters
- **return code** needs to be returned to application
  - *negative*: error code
  - *positive + 0*: success
  - usually returned via A+D registers

System call handler

- implements the actual service called through a syscall:
  1. saves tainted registers
  2. reads passed parameters
  3. sanitizes/checks parameters
  4. checks if caller has enough permissions to perform the requested action
  5. performs requested action in behalf of the caller
  6. returns to caller with success/error code

Process API — creation

- process creation events:
  1. system initialization
  2. process creation syscall
  3. user requests process creation
  4. batch job-initiation
- events map to two mechanisms:
  1. Kernel spawns initial user space process on boot (Linux: `init`)
  2. User space processes can spawn other processes (within their quota)

Process API — creation (POSIX)

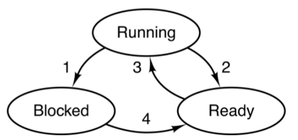
- `PID`: identifies process
- `pid = fork()`: duplicates current process:
  - returns 0 to new child
  - returns new `PID` to parent
  - ⇝ child and parent independent after `fork`
- `exec(name)`: replaces own memory based on executable file
  - `name` specifies binary executable file
- `exit(status)`: terminates process, returns `status`
- `pid = waitpid(pid, &status)`: wait for child termination
  - `pid`: process to wait for
  - `status`: points to data structure that returns information about the process (e.g., exit status)
  - passed `pid` is returned on success, `-1` otherwise
- **process tree**: processes create child processes, which create child processes, …
  - parent and child execute concurrently
  - parent waits for child to terminate (collecting the exit state)

Daemons

= program designed to run in the background
- detached from parent process after creation, reattached to process tree root (`init`)

Process States

- **blocking**: process does nothing but wait
  - usually happens on syscalls (OS doesn't run process until event happens)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process Termination

- different termination events:
  1. normal exit (voluntary)
     - `return 0` at end of `main`
     - `exit(0)`
  2. error exit (voluntary)
     - `return x` ($x \neq 0$) at end of `main`
     - `exit(x)` ($x \neq 0$)
     - `abort()`
  3. fatal error (involuntary)
     - OS kills process after exception
     - process exceeds allowed resources
  4. killed by another process (involuntary)
     - another process sends kill signal (only as parent process or administrator)

Exit Status

- voluntary exit: process returns exit status (integer)
- resources not completely freed after process terminates ⇝ **Zombie** or **process stub** (contains exit status until collected via `waitpid`)
- **Orphans**: Processes without parents
  - usually adopted by `init`
  - some systems kill all children when parent is killed
- exit status on involuntary exit:
  - Bits `0-6`: signal number that killed process (`0` on normal exit)
  - Bit `7`: set if process was killed by signal
  - Bits `8-15`: `0` if killed by signal (exit status on normal exit)

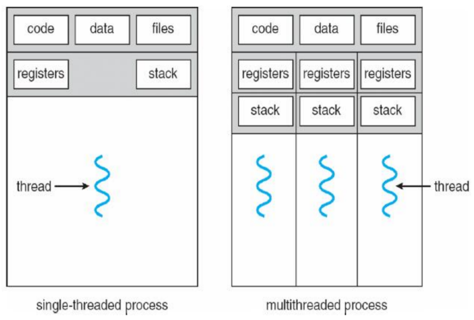
Threads

Processes vs. Threads

- **Traditional OS**: each process has
  - own address space
  - own set of allocated resources
  - *one* thread of execution (= one execution state)
- **Modern OS**: processes + threads of execution handled more flexibly
  - *processes* provide abstraction of address space and resources
  - *threads* provide abstraction of execution states of that address space
- **Exceptions**:
  - sometimes different threads have different address spaces
  - *Linux*: threads = regular processes with shared resources and AS regions

Threads — why?

- many programs do multiple things at once (e.g. web server)
  - ⇝ writing program as many sequential threads may be easier than with blocking operations
- **Processes**: rarely share data (if, then explicitly)
- **Threads**: closely related, share data



Threads — POSIX

- `PThread`: base object with
  - *identifier* (thread ID, TID)
  - *register set* (including IP and SP)
  - *stack area* to hold execution state
- `Pthread_create`: create new thread
  - Pass: *pointer* to `pthread_t` (will hold TID after successful call)
  - Pass: *attributes, start function, arguments*
  - Returns: `0` on success, error value else
- `Pthread_exit`: terminate calling thread
  - Pass: exit code (casted to void pointer)
  - Free's resources (e.g. stack)
- `Pthread_join`: wait for specified thread to exit
  - Pass: `ptread_t` to wait for (or `-1` for any thread)
  - Pass: pointer to pointer for exit code
  - Returns: `0` on success, error value else
- `Pthread_yield`: release CPU to let another thread run

Threads — Problems

- **Processes vs. Threads**:
  - *Processes*: only share resources explicitly
  - *Threads*: more shared state → more can go wrong
- **Challenges**: programmer needs to take care of
  - *activities*: dividing, ordering, balancing
  - *data*: dividing
  - *shared data*: access synchronizing

PCB vs. TCP

- **PCB** (*process control block*): information needed to implement processes
  - always known to OS
- **TCB** (*thread control block*): per thread data
  - OS knowledge depends on *thread model*

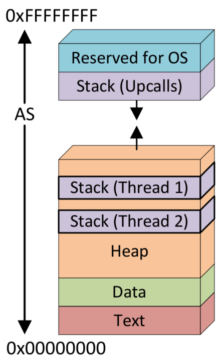
PCB	TCB
Address space	Instruction pointer
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	

Thread models

- **Kernel Thread**: known to OS kernel
- **User Thread**: known to process
- **N:1-Model**: kernel only knows one of possibly multiple threads
  - N:1 user threads = *user level threads* (ULT)
- **1:1-Model**: each user thread maps to one kernel thread
  - 1:1 user threads = *kernel level threads* (KLT)
- **M:N-Model** (hybrid model): flexible mapping of user threads to less kernel threads

Thread models — N:1

- Kernel only manages process → multiple threads unknown to kernel
- Threads managed in user-space library (e.g. GNU Portable Threads)
- **Pro**:
  + faster thread management operations (up to 100 times)
  + flexible scheduling policy
  + few system resources
  + usable even if OS doesn't support threads
- **Con**:
  - no parallel execution
  - whole process blocks if one user thread blocks
  - reimplementing OS parts (e.g. scheduler)
- **Stack**:
  - main stack known to OS used by thread library
  - own execution state (= stack) dynamically allocated by user thread library for each thread
  - possibly own stack for each exception handler
- **Heap**:
  - concurrent heap use possible
  - *Attention*: not all heaps are reentrant
- **Data**: divided into BSS, data and read-only data here as well

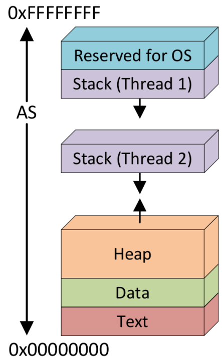


Thread models — 1:1

- kernel knows + manages every thread
- **Pros**:
  + real parallelism possible
  + threads block individually
- **Cons**:
  - OS manages every thread in system (TCB, stacks,...)
  - Syscalls needed for thread management
  - scheduling fixed in OS
- **Stack**:
  - own execution state (= stack) for every thread
  - possibly own stack for (each) exception handler
- **Heap**:
  - parallel heap use possible
  - *Attention*: not all heaps are thread-safe
  - if thread-safe: not all heap implementations perform well with many threads
- **Data**: divided into BSS, data and read-only data here as well

Thread models — M:N

- **Principle**: M ULTs are maps to (at most) N KLT
  - *Goal*: pros of ULT and KLT — non-blocking with quick management
  - create sufficient number of KLTs and flexibly allocate ULTs to them



  - *Idea*: if ULT blocks ULTs can be switched in userspace
- **Pros**:
  + flexible scheduling policy
  + efficient execution
- **Cons**:
  - hard to debug
  - hard to implement (e.g. blocking, number of KLTs,...)
- **Implementation — Up-calls**:
  - kernel notices that thread will block → sends signal to process
  - up-call notifies process of thread id and event that happened
  - exception handler of process schedules a different process thread
  - kernel later informs process that blocking event finished via other up-call

Summary

- programs often do closely related things at once
  - mapped to thread abstraction: multiple threads of execution operate in same process
- differentiation between process information (PCB) and thread information (TCB)
- **thread models**:
  - $N : 1$: threads fully managed in user-space
  - $1 : 1$: threads fully managed by kernel
  - $M : N$: threads are flexibly managed either in user-space or kernel
- multi-threaded programs operate on same data concurrently or even parallel:
  - *synchronization*: accessing such data must be synchronized
  - → makes writing such programs challenging

Scheduling

Motivation

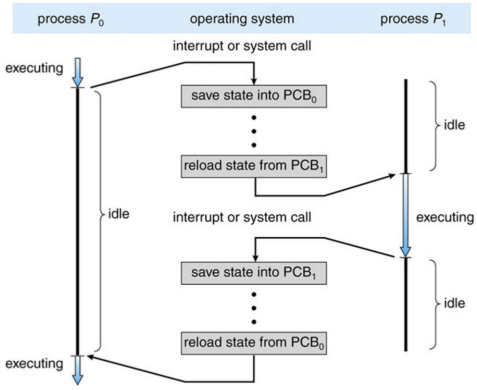
- K jobs ready to run, $K > N \geq 1$ CPUs available
- **Scheduling Problem**:
  - Which jobs should kernel assign to which CPUs?
  - When should it make decision?

Dispatcher

- **Dispatcher**: performs actual process switch
  - mechanism
  - save/restore process context
  - switch to user mode
- **Scheduler**: selects next process to run based on *policy*

Voluntary Yielding vs. Preemption

- kernel responsible for CPU switch
- kernel doesn't always run → can only dispatch different process when invoked
- **cooperative multitasking**: running process performs *yield* syscall
  → kernel switches process
- **preemptive scheduling**:
  - kernel invoked in certain time intervals
  - kernel makes scheduling decisions after every time-slice



Scheduling — Process States

- **new**: process was created but did not run yet
- **running**: instructions are currently being executed
- **waiting**: process is waiting for some event
- **ready**: process is waiting to by assigned a processor
- **terminated**: process has finished execution

Scheduling — long-term vs. short-term

- **Short-term scheduler** (CPU Scheduler, focused on in this lecture):
  - selects process to run next, allocates CPU
  - invoked frequently (ms) ⇝ must be fast
- **Long-term scheduler** (job scheduler):
  - selects process to be brought into ready queue
  - invoked very infrequently (s, m) ⇝ can be slow
  - controls degree of *multiprogramming*

Scheduling queues

- **job queue**: set of all processes in system
- **ready queue**: process in main memory, ready or waiting
- **device queue**: processes waiting for I/O device

Scheduling Policies — Categories

- **batch scheduling**:
  - still widespread in business (payroll, inventory,…)
  - no users waiting for quick response
  - non-preemptive algorithms acceptable → less switches → less overhead
- **interactive scheduling**:
  - need to optimize for response time
  - preemption essential to keep processes from hogging CPU
- **real-time scheduling**:
  - guarantee job completion within time constraints
  - need to be able to plan when which process runs + how long
  - preemption not always needed

Scheduling Policies — Goals

- **General**:
  - *fairness*: give each process fair share of CPU
  - *balance*: keep all parts of system busy
- **batch scheduling**:
  - *throughput*: number of processes that complete per time unit
  - *turnaround time*: time from job submission to job completion
  - *CPU utilization*: keep CPU as busy as possible
- **interactive scheduling**:
  - *waiting time*: reduce time a process waits in waiting queue
  - *response time*: time from request to first response
- **real-time scheduling**:
  - *meeting deadlines*: finishing jobs in time
  - *predictability*: minimize jitter

Scheduling Policies — first come first served

- intuitively clear
- **Example**: 3 processes arrive at time 0 in the order P_1, P_2, P_3

Process	Burst time	Turnaround time
P_1	24	24
P_2	3	27
P_3	3	30

- ⇝ average turnaround time 27 → *can we do better?*
- **Conclusion**: if processes would arrive in order P_2, P_3, P_1, average turnaround time would be 13
  ⇝ *good scheduling can reduce turnaround time*

Scheduling Policies — shortest job first

- **Benefits**: optimal average turnaround/waiting/response time
- **Challenge**: cannot know job lengths in advance
- **Solution**: predict length of next CPU burst for each process
  ⇝ schedule process with shortest burst next
- **Burst Estimation**: *exponential averaging*
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
  (t_n: actual length of n-th CPU burst, τ_{n+1}: predicted length of next CPU burst, $0 \le \alpha \le 1$)

Process Behavior — CPU bursts

- CPU bursts exists because processes wait for I/O
- **CPU-bound processes**: spends more time doing computations
  ⇝ few very long CPU bursts
- **I/O-bound processes**: spends more time doing I/O
  ⇝ many short CPU bursts

Scheduling Policies — preemptive shortest-job-first

- SJF optimizes waiting/response time
  ⇝ what about throughput?
- **Problem**: CPU-bound jobs hold CPU until exit or I/O → *poor I/O utilization*
- **Idea**: SJF, but preempt periodically to make new scheduling decision
  - each time slice: schedule job with shortest remaining time next
  - alternatively: schedule job with shortest next CPU burst

Scheduling Policies — round robin

- **Problem**: batch schedulers suffer from starvation and don't provide fairness
- **Idea**: each process runs for small CPU time unit
  - *time quantum/time slice* length: usually 10-100ms
  - preempt processes that have not blocked by end of time slice
  - append current thread to end of run queue, run next thread
- **Caution**: time slice length needs to balance interactivity and overhead!
  → if time slice length in the area of dispatch time, 50% of CPU time wasted for process switching

Scheduling Policies — virtual round robin

- **Problem**: RR is unfair for I/O-bound jobs: they block before using up time quantum
- **Idea**: put jobs that didn't use up their quantum in additional queue
  - store share of unused time-slice
  - give those jobs additional queue priority
  - put them back into normal queue afterwards

Scheduling Policies — (strict) priority scheduling

- **Problem**: not all jobs are equally important
  ⇝ different priorities (e.g., 4)
- **Solution**: associate priority number with each process
  - RR for each priority
  - *aging*: old low priority processes get executed before new higher priority processes

Scheduling Policies — multi-level feedback queue

- **Problem**: context switching expensive
  ⇝ trade-off between interactivity and overhead?
- **Goals**:
  - higher priority for I/O jobs (usually don't use up quantum)
  - low priority for CPU jobs (rather run them longer)
- **Idea**: different queues with different priorities and time slice lengths
  - schedule queues with (static) priority scheduling
  - double time slice length in each next-lower priority
  - process to higher priority when they don't use up quantum repetitively
  - process to lower priority when they use up quantum repetitively

Scheduling Principles — priority donation

- **Problem**: Process B (higher priority) waits for process A (lower priority)
  ⇝ B has now effectively lower priority
- **Solution**: *priority donation*
  - give A priority of B as long as B waits for A
  - if C, D, E wait for B → A gets highest priority of B, C, D, E

Scheduling Policies — lottery scheduling

- issue number of lottery tickets to processes (amount depending on priority)
- amount of tickets controls average proportion of CPU for each process
- **Scheduling**: scheduler draws random number N, process with N-th ticket is executed
- processes can transfer tickets to other processes if they wait for them

Summary

- **phases**: processes have phases of communication and waiting for I/O
  → appropriate switching between processes increases computing system utilization
- **goal-based**: scheduler decides what appropriate means based on goals
  - *long-term scheduler*: degree of multiprogramming
  - *short-term scheduler*: which process to run next
- **dispatching**: only happens when OS is invoked
  - *cooperative scheduling*: currently running thread yields (syscall)
  - *preemptive scheduling*: OS is called periodically to switch threads

Inter Process Communication

Overview

- **Reasons** for cooperating processes:
  - *information sharing*: share file/data-structure in memory
  - *computation speed-up*: break large tasks in subtasks ⇝ parallel execution
  - *modularity*: divide system into collaborating modules with clean interfaces
- **IPC**: allows data exchange
  - *message passing*: explicitly send/receive information using syscalls
  - *shared memory*: physical memory region used by multiple processes/threads

IPC — message passing

- = mechanism for processes to communicate and synchronize
- message passing facilities generally provide `send` and `receive`
- **Implementations**:
  - hardware bus
  - shared memory
  - kernel memory
  - network interface card (NIC)
- **Direct messages**: processes explicitly named when exchanging messages
- **Indirect messages**: sending to/receiving from *mailboxes*
  - first communicating process creates mailbox, last destroys
  - processes can only communicate through shared mailbox

Indirect messages – synchronization

- **Blocking** (synchronous):
  - *blocking send*: sender blocks until message is received
  - *blocking receive*: receiver blocks until message is available
- **Non-blocking** (asynchronous):
  - *non-blocking send*: sender sends message, then continues
  - *non-blocking receive*: receiver receives valid message or `null`

Messaging — Buffering

- messages are *queued* using different capacities while being in-flight
- **zero capacity**: no queuing
  - *rendezvous*: sender must wait for receiver
  - message is transferred as soon as receiver becomes available → no latency/jitter
- **bounded capacity**: finite number + length of messages
  - sender can send before receiver waits for messages
  - sender must wait if link is full
- **unbounded capacity**:
  - sender never waits
  - memory may overflow → potentially large latency/jitter between `send` and `receive`

Messaging — POSIX message queues

- **create** or open existing message queue:
  `mqd_t mq_open (const char *name, int oflag);`
  - `name` ist path in file system
  - access permission controlled through file system access permission
- **send** message to message queue:
  `int mq_send (mqd_t md, const char *msg, size_t len, unsigned priority);`
- **receive** message with highest priority in message queue:
  `int mq_receive (mqd_t md, char *msg, size_t len, unsigned * priority);`
- **register** callback handler on message queue (to avoid polling):
  `int mq_notify (mqd_t md, const struct sigevent *sevp);`
- **remove** message queue:
  `int mq_unlink (const char *name);`

Shared Memory

- **Principle**: communicate through region of shared memory
  - every write to shared region is visible to all other processes
  - hardware guarantees that always most recent write is read
- **Implementation**: message passing via shared memory is application-specific
- **Problems**: using shared memory in a safe way is tricky
  - *cache coherency protocol*: makes usage with many processes/CPUs hard
  - *race conditions*: makes usage with multiple writers hard

Shared Memory — POSIX shared memory

- **create** or open existing POSIX shared memory object:
  `int shm_open (const char *name, int oflag, mod_t mode);`
- **set** size of shared memory region:
  `ftruncate (smd, size_t len);`
- **map** shared memory object to address space:
  `void* mmap (void* addr, size_t len, [...], smd, [...]);`
- **unmap** shared memory object from address space:
  `int munmap (void* addr, size_t len);`
- **destroy** shared memory object:
  `int shm_unlink (const char *name);`

Shared Memory — sequential memory consistency

= *the result of execution as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program.*

- **Model**:
  - all memory operations occur one at a time in *program order*
  - ensures write atomicity
- **Reality**: compiler and CPU re-order instructions to *execution order*
  → without SC many processes on many CPU behave worse than preemptive threads on 1 CPU

Shared Memory — memory consistency model

- **Problem**:
  - CPUs generally not sequentially consistent
  - compilers do not generate code in program order

Synchronization — race conditions

- **Assume**: sequential memory consistency → no atomic memory transactions!
- **Critical Sections**: protect instructions inside critical section from concurrent execution

Critical Sections — desired properties

- **mutual exclusion**: at most one thread can be in the CS at any time
- **progress**: no thread running outside of CS may block other thread from getting in
- **bounded waiting**: once a thread starts trying to enter CS, there is a bound on number of times other threads get in

Critical Sections — disabling interrupts

- kernel only switches on interrupts (usually on *timer interrupt*)
  → have per-thread *do not interrupt* (DNI)-bit
- **single-core system**:

- enter CS: set DNI bit
- leave CS: clear DNI bit

- **Advantages**:
  + easy + convenient in kernel
- **Disadvantages**:
  - *only works on single-core systems*: disabling interrupts on one CPU doesn't affect other CPUs
  - *only feasible in kernel*: don't want to give user power to turn off interrupts!

Critical Sections — lock variables

- define global `lock` variable
  - only enter CS if `lock` is `0`, set to `1` on enter
  - wait for lock to become `0` otherwise (*busy waiting*)
- **Problem**: doesn't solve CS problem! Reading/Setting lock not atomic!

Critical Sections — spinlocks

- to make lock variable approach work, lock variable must be tested and set at same time atomically:
- **x86**: `xchg` can atomically exchange memory content with register
  - exchanges register content with memory content
  - returns previous memory content of lock
- ⇝ implementation of critical section as *spinlock*:

```
void enter_critical_section (volatile bool *lock) {
  while (xchg(lock, 1) == 1); // lock = 1, return old value
                              // repeat until old value != 1
}

void leave_critical_section (volatile bool *lock) {
  *lock = 0;
}
```

- **Advantages**:
  + *mutual exclusion*: only one thread can enter CS
  + *progress*: only thread within CS hinders others of getting in
- **Disadvantages**:
  - *bounded waiting*: no upper bound

Spinlocks — Limitations

- **Congestion**:
  - if most times there is no thread in CS when another tries to enter, then spinlocks are very easy + efficient
  - if CS is large or many threads try to enter, spinlocks might not be good choice as all threads actively wait spinning
- **Multicore**: memory address is written at every atomic swap operation
- → memory is expensively kept coherent
- **Static Priorities** (e.g., *priority inversion*): if low-priority threads hold lock it will never be able to release it, because it will never be scheduled

Spinlocks — sleep while wait

- **Problem**: busy part of busy waiting
  - wastes resources,
  - stresses cache coherence protocol,
  - can cause priority inversion problem
- **Idea**:
  - threads sleep on locks if occupied
  - wake up threads one at a time when lock becomes free

Spinlocks — semaphore

- two new syscalls operating on `int` variables:
  - `wait (&s)`: if `s > 0`: `s--` and continue, otherwise let caller sleep
  - `signal (&s)`: if no thread is waiting: `s++`, otherwise wake one up
- initialize `s` to maximum number of threads that may enter CS
  - `wait` = `enter_critical_section()`
  - `signal` = `leave_critical_section()`
- **mutex** (semaphore): semaphore initialized to 1 (only admits one thread at a time into CS)
- **counting semaphore**: semaphore allowing more than one thread into CS at a time

Semaphore — implementation

- `wait` and `signal` calls need to be carefully synchronized (otherwise *race condition* between checking and decrementing `s`)
- **signal loss** can occur when waiting and waking threads up at same time
- ⇝ each semaphore has **wake-up queue**:
  - *weak semaphores*: wake up random waiting thread on `signal`
  - *strong semaphores*: wake up thread strictly in order which they started `wait`ing
- **Advantages**:
  + *mutual exclusion*: only one thread can enter CS for mutexes
  + *progress*: only thread within CS hinders others to get in
  + *bounded waiting*: strong semaphores guarantee bounded waiting
- **Disadvantages**:
  - every enter and exit of CS is syscall → slow

Fast User Space mutex

- **spinlock**:
  + quick when wait-time is short
  - waste resources when wait-time is long
- **semaphore**:
  + efficient when wait-time is long
  - syscall overhead at every operation
- **futex**:
  - userspace + kernel component
  - try to get into CS with userspace spinlock
    - CS busy → use syscall to put thread to sleep
    - otherwise → enter CS with now locked spinlock completely in userspace

Summary

- **communication** between processes/threads often needed
  - *message passing*: provide explicit send/receive functions to exchange messages
  - *implicitly/explicitly shared memory* between threads/processes: allows information exchange
- **data races**: need to be taken into account when communicating
- **synchronization techniques**:
  - interlocked atomic operations
  - spinlocks
  - semaphores
  - futexes

Synchronization and Deadlocks

Producer-Consumer Problem

- **Definition**:
  - buffer is shared between producer and consumer (LIFO)
  - `count` integer keeps track of number of currently available items
  - producer produces item → placed in buffer, `count++`
  - buffer full → producer needs to sleep until consumer consumed an item
  - consumer consumes item → remove item from buffer, `count--`
  - buffer empty → consumer needs to sleep until producer produces item
- **Problem**: *race condition* on `count`

Producer-Consumer Problem — condition variables

- **Solution**: can be solved with mutex + 2 counting semaphores
  - hard to understand
  - hard to get right
  - hard to transfer to other problems
- **condition variables**: allow blocking until condition is met
  - usually suitable for same problems but much easier to get right
- **Idea**:
  - new operation performs *unlock, sleep, lock* atomically
  - new wake-up operation is called with lock held
- → simple mutex lock/unlock around CS + no signal loss
- **Pthread** condition variables:
  - `pthread_cond_init`: create + initialize new CV
  - `pthread_cond_destroy`: destroy + free existing CV
  - `pthread_cond_wait`: block waiting for signal
  - `pthread_cond_timedwait`: block waiting for signal or timer
  - `pthread_cond_signal` : signal another thread to wake up
  - `pthread_cond_broadcast`: signal all threads to wake up

Reader-Writer Problem

- **Problem**: model access to shared data structures

```
void producer()
{
  Item newItem;

  for(;;) // ever
  {
    newItem = produce();

    mutex_lock( &lock );
    while( count == MAX_ITEMS )
      cond_wait( &less, &lock );

    insert( newItem );
    count++;

    cond_signal( &more );
    mutex_unlock( &lock );
  }
}
```

```
void consumer()
{
  Item item;

  for(;;) // ever
  {
    mutex_lock( &lock );
    while( count == 0 )
      cond_wait( &more, &lock );

    item = remove();
    count--;

    cond_signal( &less );
    mutex_unlock( &lock );

    consume( item );
  }
}
```

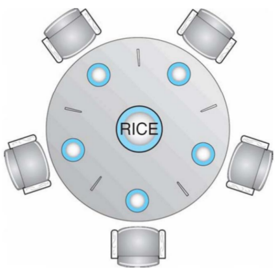
- many threads compete to read/write same data
- *readers*: only read data set, not performing any updates
- *writers*: both read and write

⇝ using single mutex for read/write operations is not a good solution! (unnecessarily blocking out multiple readers while no writer is present)

- **Idea**: locking should reflect different semantics for reading/writing
  - no writing thread → multiple readers may be present
  - writing thread → no other reader/writer allowed

Dining-Philosophers Problem

- **Definition**: 5 philosophers with cyclic workflow:
  1. think
  2. get hungry
  3. grab one chopstick
  4. grab other chopstick
  5. put down chopsticks
- **Rules**:
  - no communication
  - no atomic grabbing of both chopsticks
  - no wrestling
- **Abstraction**: models threads competing for limited number of resources **Problem**: what happens if all philosophers grab left chopstick at once?
- **Deadlock workarounds**:
  - *deadlock avoidance*: just 4 philosophers allowed at table of 5
  - *deadlock prevention*: odd philosophers take left chopstick first, even ones take right first → *deadlock prevention*



Deadlocks

- **Deadlocks** can arise if all four conditions hold simultaneously:
  - *mutual exclusion*: limited resource access (can only be shared with finite number of users)
  - *hold and wait*: wait for next resource while already holding at least one
  - *no preemption*: granted resource cannot be taken away but only handled back voluntarily
  - *circular wait*: possibility of circularity in requests graph

Deadlocks — countermeasures

- **prevention**: pro-active, make deadlocks impossible to occur
- **avoidance**: decide on allowed actions based on a-priori knowledge
- **detection** (*recovery*): react after deadlock happened

Deadlocks — prevention

- **Goal**: negate at least one of the required deadlock conditions:
  - *mutual exclusion*: buy more resources, split into pieces, virtualize
  - *hold and wait*: get all resources en-bloque, 2-phase-locking
  - *no preemption*: virtualize to make preemptable
  - *circular wait*: reorder resources, prevent through partial order on resources

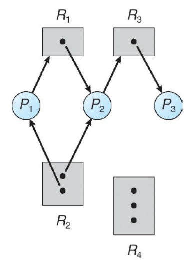
Deadlocks — avoidance

- **safe state**: system is in safe state → no deadlocks
- **unsafe state**: system is in unsafe state → deadlocks possible
- **avoidance**: on every resource request decide if system stays in safe state

→ *resource allocation graph*

Deadlock Avoidance — resource allocation graph

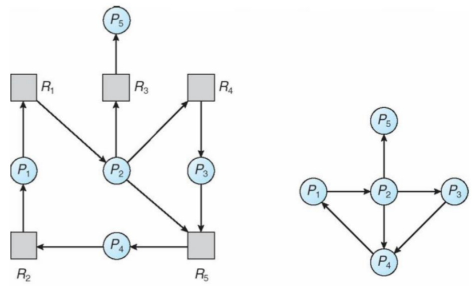
- **principle**: view system state as graph
  - *processes* = round nodes
  - *resources* = square nodes
  - *resource instance* = dot in resource node
  - *resource requests/assignments* = edges
    - resource → process = resource is assigned to process
    - process → resource = process is requesting resource



Deadlocks — detection

- **Principle**: allow system to enter deadlock → detection → recovery scheme
- **wait-for graph** (WFG):
  - *processes* = nodes
  - *wait-for relationship* = edges
- periodically invoke algorithm searching for cycle in graph

⇝ cycle exists → deadlock exists



Deadlocks — recovery

- **Process termination**:
  - *all*: abort all deadlocked processes
  - *selective*: abort one process at a time until deadlock is eliminated
- **Termination order**: in which order should processes be aborted?
  - process priority
  - how long already computed? how much longer for completion?
  - amount of resources used
  - amount of resources needed for completion
  - how many processes will need to be terminated
  - interactive or batch?
- **Resource preemption**:
  - *victim selection*: minimize cost
  - *rollback*: perform periodic snapshots, abort process to preempt resources → restart from last safe state
  - *starvation*: same process may always be picked as victim ⇝ include rollback count in cost factor

Summary

- **classical synchronization problems**: model synchronization problems occurring in reality
  - *producer-consumer*: shared use of buffers/queues
  - *reader-writer*: shared access to data structures
  - *dining philosophers*: competition for limited resources
- such synchronization problems occur very often when programming operating systems
- **parallelism**: introduced by multiple processors + multiprogramming, needs to be considered carefully when writing OS

Memory Management Hardware

Main Memory

- main memory + registers = only storage that CPU can access directly
- **Before run**: program must be
  - brought into memory from background storage
  - placed within a process' address space
- **Earlier**: computers had no memory abstraction
  → programs accessed physical memory directly
- multiple processes can be run concurrently even without memory abstraction (using swapping, relocation)

Swapping

- **Principle**:
  - *roll-out*: save program's state on background storage
  - *roll-in*: replace program state with another program's state
- **Advantages**:
  + only needs hardware support to protect kernel, not to protect processes from one another
- **Disadvantages**:
  - *very slow*: major part of swap time is transfer time
  - *no parallelism*: only one process runs at a time, owns entire physical address space

Overlays

- **Problem**: what if process needs more memory than available?
  → need to partition program manually

Static Relocation

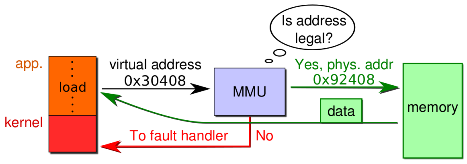
= OS adds fixed offset to every address in a program when loading + creating process
- same address space for every process
  → *no protection*: every program sees + can access every address!

Shared Physical Memory — Goals

- **Protection**:
  - bug in one process must not corrupt memory in another
  - do not allow processes to observe other processes' memory
- **Transparency**:
  - process should not require particular physical memory addresses
  - processes should not be able to use large amounts of contiguous memory
- **Resource Exhaustion**: allow that sum of sizes of all processes is greater than physical memory

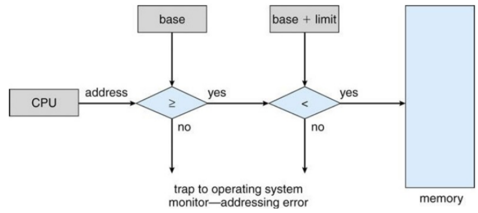
Memory Management Unit

- **Motivation**: need hardware support to achieve safe + secure protection
- **Goal**: hardware maps virtual to physical address
- **Usage**: user program deals with virtual addresses, never sees real addresses



MMU — base and limit registers

- **Idea**: provide protection + dynamic relocation in MMU
  → introduce special *base* and *limit* registers (e.g., Cray-1)
- **Usage**: on every load/store the MMU
  - checks if virtual address ≥ `base`
  - checks if virtual address < `base` + `limit`
  - use virtual address as physical address in memory
- **Protection**: OS needs to be protected from processes
  - main memory split in two partitions (low = OS, high = user processes)
  - OS can access all process partitions (e.g., to copy syscall parameters)
  - MMU denies processes access to OS memory
- **Advantages**:
  + straight forward to implement MMU
  + very quick at run-time
- **Disadvantages**:
  + how to grow process' address space?
  + how to share code/data?

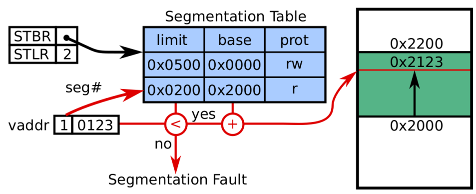


MMU — Segmentation

- **Solution** to base + limit: use multiple base + limit register pairs *per process*
  → private + public segments
- **Advantages**:
  + data/code sharing between processes possible without compromising confidentiality
  + process does not need large contiguous physical memory area → easy placement
  + process does not need to be entirely in memory → memory overcommitment ok
- **Disadvantages**:
  - segments need to be kept contiguous in physical memory
  - *fragmentation* of physical memory

Segmentation — Architecture

- virtual address = [segment #, offset]
- each process has *segment table*, maps virtual address to physical address in memory
  - *base*: starting physical address where segment resides in memory
  - *limit*: length of segment
  - *protection*: access restriction (read/write) for safe sharing
- MMU has two registers that identify current address space
  - *segment-table base register* (STBR): points to segment table location of current process
  - *segment-table length register* (STLR): indicates number of segments used by process



External Fragmentation

- **Fragmentation** = inability to use free memory
- **External Fragmentation** = sum of free memory satisfies requested amount of memory, but is not contiguous
- **Compaction**: reduce external fragmentation
  - close gaps by moving allocated memory in one direction
  - only possible if relocation is dynamic, can be done at execution time
  - *problem*: expensive! Need to halt process while moving data and updating tables
    → caches need to be reloaded, which should be avoided

MMU — Paging

- **Principle**: divide physical memory into fixed-size blocks (*page frames*)
  - size = 2^n Bytes (typically 4KiB, 2MiB, 4MiB)
- **Virtual Memory**: divided into same-sized blocks (*pages*)
- **Page Table**: managed by OS, stores mappings between *virtual page numbers* (vpn) and *page frame numbers* (pfn) for each AS
- OS tracks all free frames, modifies page tables as needed
- **Present Bit** (in page table): indicates that virtual page is currently mapped to physical memory
- if process issues instruction to access unmapped virtual address, MMU calls OS to bring in the data (*page fault*)

MMU — Address Translation Scheme

- **Virtual address**: divided into
  - *virtual page number*: page table index containing base address of each page in physical memory
  - *page offset*: concatenated with base address results in physical address

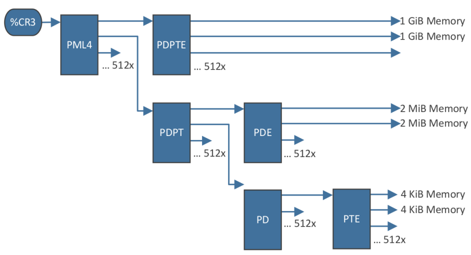
MMU — Hierarchical Page Table

- **Problem**: need to keep complete page table in memory for every address space
- **Idea**: not needing complete table, most virtual addresses unused by process

→ subdivide virtual address further into multiple page indexes p_n forming *hierarchical page table*

Hierarchical Page Table — x86-64

- **long mode**: 4-level hierarchical page table
- **page directory base register** (control register 3, `%CR3`) stores starting physical address of *first level page table*
- **address-space hierarchy**: following page-table hierarchy for every address space:
  - page map level 4 (PML4)
  - page directory pointers table (PDPT)
  - page directory (PD)
  - page table entry (PTE)
- **Per level**: table can either point to *directory* in next hierarchy level or to *entry* containing actual mapping data



Page Table Entry — Content

- **valid bit** (*present bit*): whether page is currently available in memory or needs to be brought in by OS via *page fault*
- **page frame number**: if page is present: physical address where page is currently located
- **write bit**: whether or not page may be written to (may cause *page fault*)
- **caching**: whether or not page should be cached at all (and with which policy)
- **accessed bit**: set by MMU if page was touched since bit was last cleared by OS
- **dirty bit**: set by MMU if page was modified since bit was last cleared by OS

Paging — OS Involvement

- OS performs all operations that require semantic knowledge
- **page allocation** (bringing data into memory): OS needs to find free frame for new pages and set up mapping in page table of affected address space
- **page replacement**: when all page frames are used, OS needs to evict pages from memory
- **context switching**: OS sets MMU's base register (`%CR3` on x86) to point to page hierarchy of next process's address space

MMU — Internal Fragmentation

- **Paging**: eliminates external fragmentation
- **Problem**: internal fragmentation
  - memory can only be allocated in page frame sizes
  - allocated virtual memory area will generally not end at page boundary

⇝ unused rest of last allocated page is lost!

MMU — Page Size trade-offs

- **Fragmentation**:
  - *larger pages* → more memory wasted (internal fragmentation) per allocation
  - *smaller pages* → only half a page wasted per allocation on average
- **Table Size**:
  - *larger pages* → fewer bits needed for `pfn` (more bits in offset), fewer PTEs
  - *smaller pages* → more + larger PTEs
- **I/O**:
  - *larger pages* → more data needs to be loaded from dist to make page valid
  - *smaller pages* → need to trap OS more often when loading large program

Summary

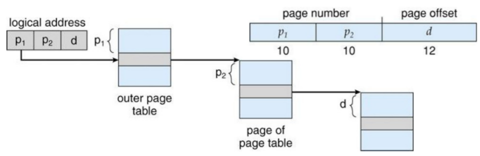
- need to place processes in memory to run
- want to place multiple processes in memory at same time to run concurrently/parallel
- **Virtual Memory**: enables protection, transparency, overcommitment
  - emphtrade-off extra hardware (MMU) to translate addresses at every load-/store
- **MMU types**: base + limit, segmentation, paging
- **Paging**: supported by all contemporary MMUs, favorite of most OS

Paging

Hierarchical Page Table — two-level page table

- **Layout**: on 32-bit machine with 4KiB pages divide virtual address into
  - *page number* (p): 20 bits
  - *page offset* (d): 12 bits
- **Table Paging**: table can be paged to save memory – subdivide vpn:
  - index in *page directory* (p_1): 10 bits
  - index in *page table entry* (p_2): 10 bits
- for ranges of 1024 invalid pages, reset present bit in page directory

→ save space of second-level page table



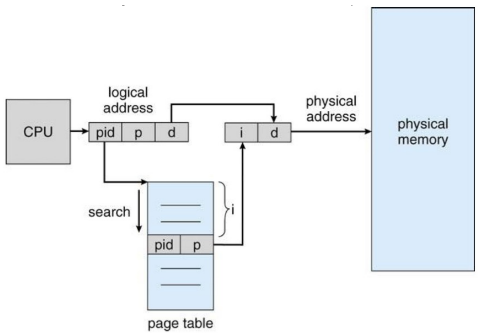
Linear Inverted Page Table

- **Problem**: large AS (64 bit) but only few mapped virtual addresses

→ much memory wasted on page tables

→ lookup slow due to many levels of hierarchy

- **Idea**: invert page table mapping
  - map physical frame to virtual page instead of other way around
  - single page table for *all processes* (exactly one table per system)
  - one page table entry for each physical page frame
- **Advantage**: less overhead for page table meta data
- **Disadvantage**: increases time needed to search table when page reference occurs



Hashed Inverted Page Table

- **Hash Anchor Table**: limits search to at most a few page-table entries

Translation Lookaside Buffer — Motivation

- **Naive paging is slow**:
  - every load/store requires multiple memory references
  - 4-level hierarchy: 5 memory references for every load/store (4 page directory/table references, 1 data access)
- **Idea**: add cache that stores recent memory translations
  - *translation lookaside buffer* (TLB) maps [vpn] to [pfn, protection]
  - typically 4-way to fully associative hardware cache in MMU
  - typically 64-2000 entries
  - typically 95%-99% hit rate

TLB — Operation

- on every load/store:
  - check if translation result is cached in TLB (*TLB hit*)
  - otherwise walk page tables, insert result into TLB (*TLB miss*)
- **Quick**: can compare many TLB entries in parallel in hardware

TLB — TLB Miss

- **Process**:
  - evict entry from TLB on TLB miss
  - load entry for missing virtual address into TLB
- **Variants**: *software-managed* and *hardware managed*
- **software-managed TLB**:
  - OS receives *TLB miss exception*
  - OS decides which entry to evict (drop) from TLB
  - OS generally walks page tables in software to fill new TLB entry
  - TLB entry format specified in *instruction set architecture* (ISA)
- **hardware-managed TLB**:
  - evict TLB entry based on hardware-encoded policy
  - walk page table in hardware → resolve address mapping

TLB — Address Space Identifiers

- **Problem**: vpn dependent on AS
  - vpns in different AS can map to different pfns
  - → need to clear TLB on AS switch
- **Idea**: solve vpn ambiguity with additional identifiers in TLB
- **ASID**: TLB has *address space identifier* (ASID) in every entry
  - map [vpn, ASID] to [pfn, protection]
  - → avoids TLB flush at every address-space switch
  - → less TLB misses: some TLB entries still present from last time process ran

TLB — Reach

= amount of memory accessible with TLB hits: TLB reach = TLB size ∗ page size

- **Ideally**: working set of each process is stored in TLB (otherwise high degree of TLB misses)
- **Increase page size**:
  - \+ fewer TLB entries per memory needed
  - \- increase internal fragmentation
- **multiple page sizes**:
  - \+ allows applications that map larger memory areas to increase TLB coverage with minimal fragmentation increase
- **increase TLB size**:
  - \- expensive

TLB — Effective Access Time

- **Associative lookup**: takes τ time units (e.g., $\tau = 1$ns)
- **Memory cycle**: takes μ time units (e.g., $\mu = 100$ns)
- **TLB hit ratio** α: percentage of all memory accesses with cached translation (e.g., $\alpha = 99\%$)
- **Effective Access Time** (EAT) for linear page table without cache:

$$\text{EAT} = (\tau + \mu)\alpha + (\tau + 2\mu)(1 - \alpha) = \tau + 2\mu - \mu\alpha$$

Summary

- page tables communicate between OS and MMU hardware
  - how virtual addresses in each address space translate to physical addresses
  - which kind of accesses the MMU should allow/signal to the OS
- different page table layouts have been developed
  - linear page table
  - hierarchical page tables
  - inverted page tables
  - hashed page tables
- performing page table lookups for every memory access significantly slows down execution time of programs
  - translation lookaside buffer (TLB) caches previously performed page table lookups
  - typical TLBs cover $95\% - 99\%$ of all translations

Caching

Caching — Motivation

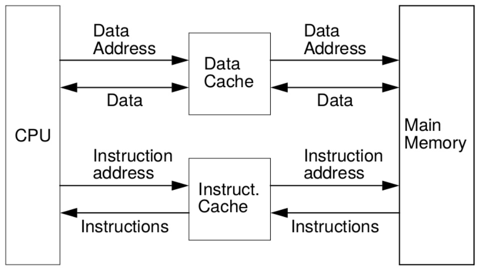
- memory (RAM) needs to be managed carefully
- **Ideal** properties: large, fast, nonvolatile, cheap
- **Real** memory: trade-offs

Caching — Cache misses

- **Compulsory miss**:
  - cold start, first reference
  - data block was not cached before
- **Capacity miss**:
  - not all required data fits into cache
  - accessed data previously evicted to make room for different data
- **Conflict miss**:
  - collision, interference
  - depending on cache organization, data items interfere with each other
  - fully associative caches are not prone to conflict misses

Caching — Harvard architecture

- **Principle**: separate buffer memory for data and instructions



Caching — write/replacement policies

- **Cache hit**:
  - *write-through*: main memory always up-to-date, writes might be slow
  - *write-back*: data written only to cache, main memory temporarily inconsistent
- **Cache miss**:
  - *write-allocate*: data read from main memory to cache, write performed afterwards
  - *write-to-memory*: modification is performed only in main memory

Cache Design Parameters

- **Size** + **Set size**: small cache → set-associative implementation with large sets
- **Line length**: spatial locality → long cache lines
- **Write policy**: temporal locality → write-back
- **Replacement policy**
- **Tagging/Indexing**: virtual or physical addresses

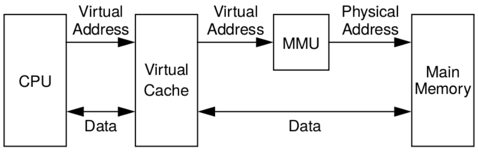
Caching — Problems

- **Ambiguity problem**: same virtual addresses point to different physical addresses at different times
- **Alias problem**: different virtual addresses point to same physical memory location

Caching — virtually indexed, virtually tagged

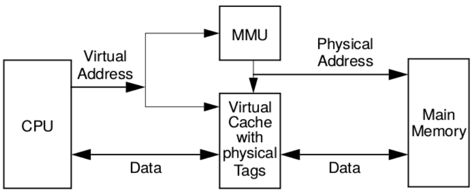
- **Operations**:
  - *context switch*: cache must be invalidated (and written back if write-back is used)
  - *fork*: child needs complete copy of parent's address space
  - *exec*: invalidate cache, no write-back necessary
  - *exit*: flush cache
  - *brk/sbrk*: growing = nothing, shrinking = (selective) cache invalidations
- **shared memory/memory-mapped files**: alias problem!
  - disallow, do not cache
  - only allow addresses mapping to same cache line (if using direct-mapped write-allocate cache)
  - each frame accessible from exactly one virtual address at any time → alias page invalidation
- **I/O**:
  - *buffered I/O*: no problems

  - *unbuffered I/O*:
    - write: information may still be in cache → write back before I/O starts
    - read: cache must be invalidated



Caching — virtually indexed, physically tagged

- **Usage**: often used as first-level cache
- **Management**:
  - no ambiguities
  - no cache flush/context switch
  - shared memory/memory mapped files: virtual starting addresses must be mapped to same cache line
  - I/O: cache flush required
- **Conflicts**: data structures with address distance = multiple of cache size are mapped to same line
- **Runtime properties**:
  - *cache flush*: avoidable most times (fast context switches, interrupt-handling, syscalls)
  - *deferred write-back after context switch*:
    - avoids write accesses → performance gain
    - variable execution time caused by compulsory misses
  - *dynamic memory management*: causes variable execution times through conflict misses
  - *multiprocessor systems*: problematic with shared memory — which line should be invalidated?
    - cache size is small multiple of page size (1-4)
    - requires to only invalidate/flush 1-4 cache lines by cache coherency HW



Caching — physically indexed, physically tagged

- **Advantages**:
  - \+ completely transparent to processor
  - \+ no performance-critical system support required (including I/O)
  - \+ SMPs with shared memory can use coherency protocol implemented in hardware
- **random allocation conflicts**:
  - page conflicts caused by random allocation of physical memory
  - contiguous virtual memory normally mapped to arbitrary free physical pages
- **random coloring conflicts**: consequences of random page coloring:
  - cache conflicts
  - cache only partially used
  - significant runtime variations
- **conflict mitigation**:
  - sequential page colors for individual memory segments
  - *cache partitioning*: divide physical memory in disjoint subsets, all pages of subset are mapped to same cache partition